Algorithms and Data Structures 2
CS 1501

Spring 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming deadlines:
  - Homework 12 due on 4/18
  - Assignment 3 and 4 due on 4/18
  - Lab 12 due on 4/22
  - Assignment 5 due on 5/2
  - Bonus Opportunities:
    - Bonus Lab due on 5/2
    - Bonus Homework due on 5/2
    - 1 bonus point for entire class when response rate >= 80%
      - Currently at ~11%
      - Deadline is Sunday 4/24

Previous lecture …

- (Big)Integer Algorithms
  - exponentiation
  - GCD
  - Random generation of large prime numbers
- RSA Security

CourseMIRROR Reflections (most confusing)

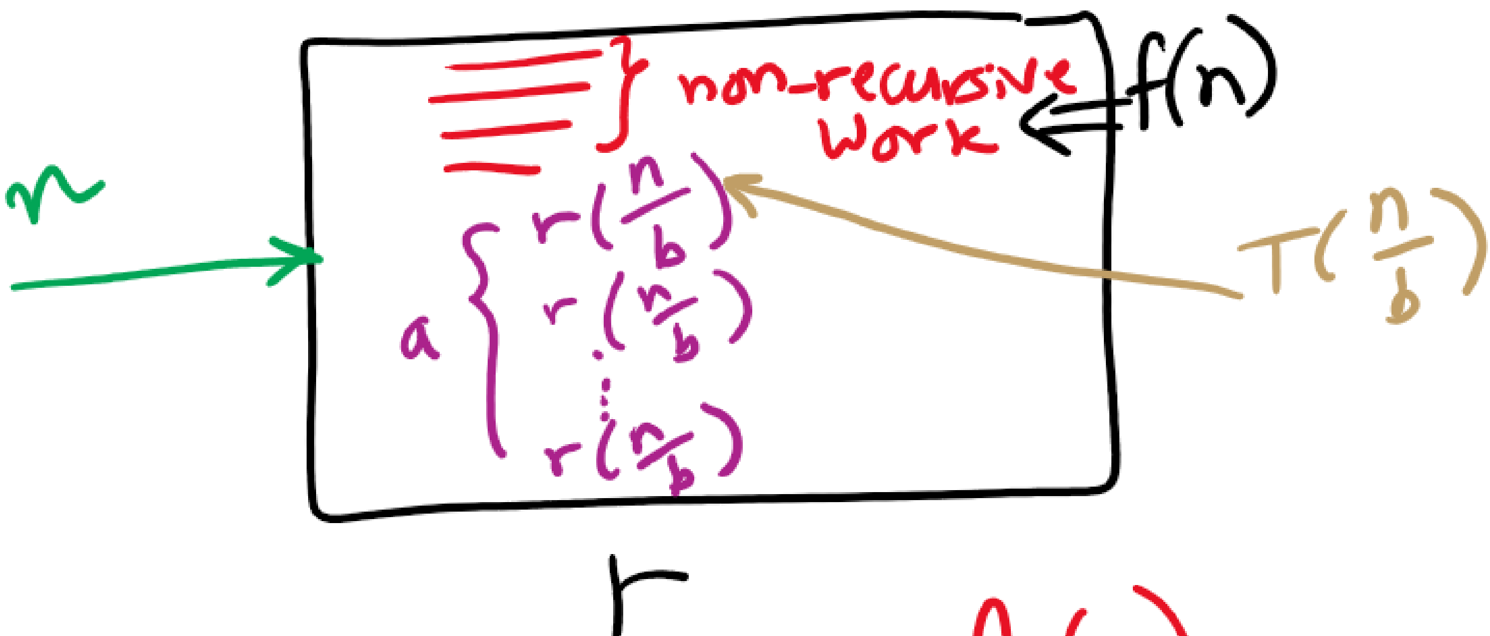
- Symmetric vs asymmetric encryption, what is the role of keys in both situations?
- what is the purpose of euclids algorithm and extended euclids algorithm?
- why gcd is exponential runtime but also linear runtime
- I would like to go over more examples with GCD
- Extended Euclids Algorithm was a bit fast and Im not sure I understand it
- I was confused about the hashing example
- I was confused about the example of using a hash function with RSA
- I was confused how signing a message with RSA encryption worked
- how rsa works was a little confusing
- RSA keypairs
- Lots of number theory very fast. Will take time to digest. Not connected to class: why will quantum computers make rsa no longer useful?
- Recurrence relations were most confusing.
- The master method. We kind of ran out of time on it. Would like to see more

CourseMIRROR Reflections (most interesting)

- Different ways of encryption through algorithms/encryption in general
- Simplicity of effective encryption techniques
- How simple encoding can be sometimes
- use of one time pads in encryption
- Using algorithms from thousands of years ago to use stuff we use daily
- I found the idea of Eulers totient to be an interesting idea, I wonder what other properties it has
- Euclids method was interesting especially how it was faster than brute force
- Euclids algorithm was very genius
- the rsa encrypt and decrypt example
- Learning about RSA envelopes and how they're used everywhere
- using an RSA envelope to cut down on runtime
- The giant runtimes of some of the example algorithms we went over
- I found the time to decrypt in years for some parts very interesting
- The use of modular arithmetic to simplify exponentiation in RSA
- I enjoy learning about crypto

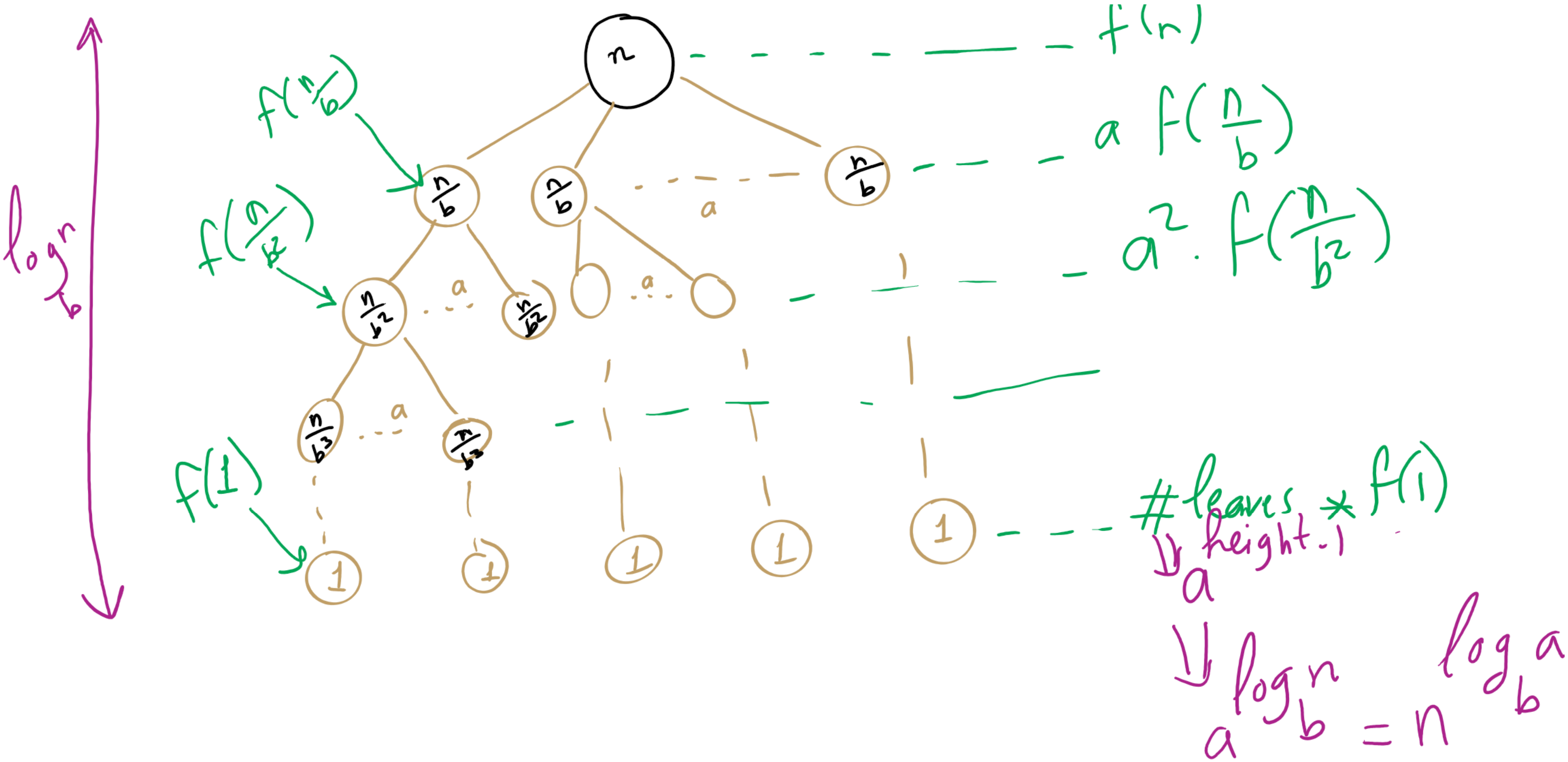
Run-time Analysis of Recursive Algorithms

$T(n)$: runtime



$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

Recursion Tree



Applying the master theorem

$$T(n) = aT(n/b) + f(n)$$

- If $f(n)$ is $O(n^{\log\_b(a) - \varepsilon})$:
  - $T(n)$ is $\Theta(n^{\log\_b(a)})$
- If $f(n)$ is $\Theta(n^{\log\_b(a)})$
  - $T(n)$ is $\Theta(n^{\log\_b(a)} \lg n)$
- If $f(n)$ is $\Omega(n^{\log\_b(a) + \varepsilon})$ and $(a * f(n/b) <= c * f(n))$ for some $c < 1$:
  - $T(n)$ is $\Theta(f(n))$

The 3 cases of the Master Theorem

$f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ $\quad a \cdot f\left(\frac{n}{b}\right) < c \cdot f(n)$ $\quad c < 1$ $\quad \boxed{\text{Case 3}}$ $\quad T(n) = \Theta(f(n))$

$f(n) = \Theta\left(n^{\log_b a}\right)$ $\quad \boxed{\text{Case 2}}$ $\quad T(n) = \log_b n * f(n) = \log_b n * n^{\log_b a}$

work at root $\times$ # levels

$f(n) = O\left(n^{\log_b a - \epsilon}\right)$ $\quad \boxed{\text{Case 1}}$ $\quad T(n) = \Theta\left(n^{\log_b a}\right)$

$\Theta(\text{leaves})$

Mergesort master theorem analysis

Recurrence relation for mergesort? $T(n) = 2T(n/2) + \Theta(n)$

- $a = 2$
- $b = 2$
- $f(n)$ is $\Theta(n)$
- So…
  - $n^{\log\_b(a)} = \ldots$
    - $n^{\lg 2} = n$

- If $f(n)$ is $O(n^{\log\_b(a) - \varepsilon})$:
  - $T(n)$ is $\Theta(n^{\log\_b(a)})$
- If $f(n)$ is $\Theta(n^{\log\_b(a)})$
  - $T(n)$ is $\Theta(n^{\log\_b(a)} \lg n)$
- If $f(n)$ is $\Omega(n^{\log\_b(a) + \varepsilon})$

  and $(a * f(n/b) <= c * f(n))$ for some $c < 1$:
  - $T(n)$ is $\Theta(f(n))$

  - Being $\Theta(n)$ means $f(n)$ is $\Theta(n^{\log\_b(a)})$
  - $T(n) = \Theta(n^{\log\_b(a)} \lg n) = \Theta(n^{\lg 2} \lg n) = \Theta(n \lg n)$

Binary Search Master Theorem Analysis

Binary Search

```
if a[mid] == target
    return mid
if a[mid] < target
    recurse right half
else
    recurse left half
```

$$T(n) = 1 \cdot T\left(\frac{n}{2}\right) + \Theta(1)$$

a b $f(n)$

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$$

Case 2:

$$T(n) = \log_b n * 1$$

$$= \log_2 n$$

For our divide and conquer multiplication approach

$$T(n) = 4T(n/2) + \Theta(n)$$

- $a = 4$
- $b = 2$
- $f(n)$ is $\Theta(n)$
- So…
  - $n^{\log\_b(a)} = \ldots$
    - $n^{\lg 4} = n^2$

> - If $f(n)$ is $O(n^{\log\_b(a) - \varepsilon})$:
>   - $T(n)$ is $\Theta(n^{\log\_b(a)})$
> - If $f(n)$ is $\Theta(n^{\log\_b(a)})$
>   - $T(n)$ is $\Theta(n^{\log\_b(a)} \lg n)$
> - If $f(n)$ is $\Omega(n^{\log\_b(a) + \varepsilon})$
>
>   and $(a * f(n/b) <= c * f(n))$ for some $c < 1$:
>   - $T(n)$ is $\Theta(f(n))$

- Being $\Theta(n)$ means $f(n)$ is polynomially smaller than n^2
- $T(n) = \Theta(n^{\log\_b(a)}) = \Theta(n^{\lg 4}) = \Theta(n^2)$

Attacking a Recurrence Relation to Reduce Run-time

- Look at the recurrence relation again to see where we can improve our runtime:

$$T(n) = 4T(n/2) + \Theta(n)$$

Can we reduce the number of subproblems?

Can we reduce the subproblem size?

Can we reduce the amount of work done by the current call?

Karatsuba runtime

The recurrence relation for Karatsuba's algorithm is:

- $T(n) = 3T(n/2) + \Theta(n)$
  - Which solves to be $\Theta(n^{\lg 3})$
    - Asymptotic improvement over grade school algorithm!
      - For large n, this will translate into practical improvement

n^2 $\qquad$ $n^{1.58}$

Karatsuba's

Karatsuba Runtime

$$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n)$$

a b $f(n)$

$$n^{\log_b a} = n^{\log_2 3} = n^{1.58} > f(n)$$

Polynomially

Case 1 $\quad T(n) = \Theta\left(n^{\log_b a}\right) = \Theta(n^{1.58})$

When can we use the Master Theorem?

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

1) all subproblems are of equal size

2) subproblem size is a fraction of the problem size

3) $f(n)$ → $\Theta(n^{\log_b a})$

→ Polynomially larger/smaller

When Master Theorem doesn't apply: Example 1

$$T(n) = \overset{a}{2}\,T\left(\frac{n}{\underset{b}{2}}\right) + n \log n$$

$a = 2$

$b = 2$

$f(n) = n \log n$

$$n^{\log_b a} = n^{\log_2 2} = n^1$$

$$\underline{n \log n} \cancel{\overset{?}{=}} \Omega(n^{\underline{1+\epsilon}})$$

$$n \log n < n^{1.0000001}$$

When Master Theorem doesn't apply: Example 2

- Top-down divide and conquer algorithm for exponentiation
- $x^y = (x^{(y/2)})^2 = x^{(y/2)} * x^{(y/2)}$
  - Similarly, $(x^{(y/2)})^2 * x = x^{(y/2)} * x^{(y/2)} * x$
- So, our recurrence relation is:
  - $T(n) = T(n-1) + ?$
    - How much work is done per call?
    - 1 (or 2) multiplication(s)
      - Examined runtime of multiplication last lecture
      - But how big are the operands in this case?

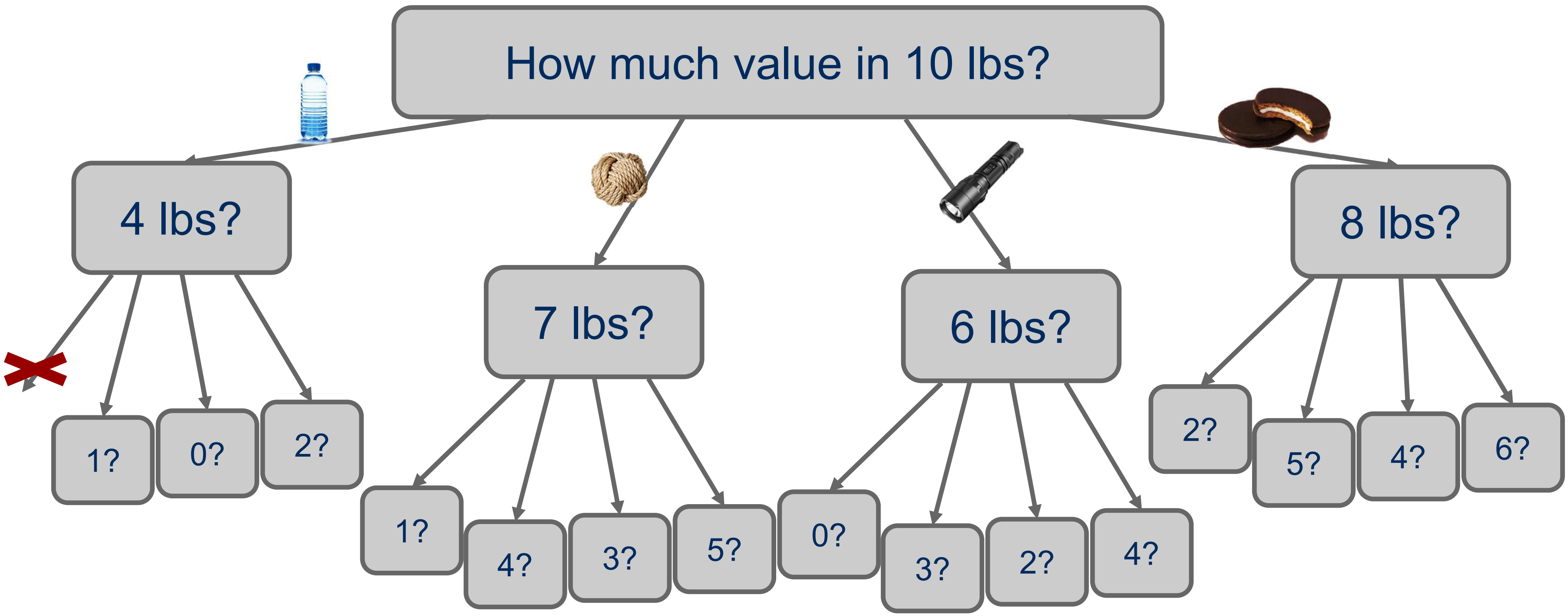
Problem of the Day Part 3: The unbounded knapsack problem

- Given a knapsack that can hold a weight limit L, and a set of n types items that each has a weight (w_i) and value (v_i), what is the maximum value we can fit in the knapsack if we assume we have unbounded copies of each item?

Recursive Solution

weight:	6	3	4	2
value:	30	14	16	9

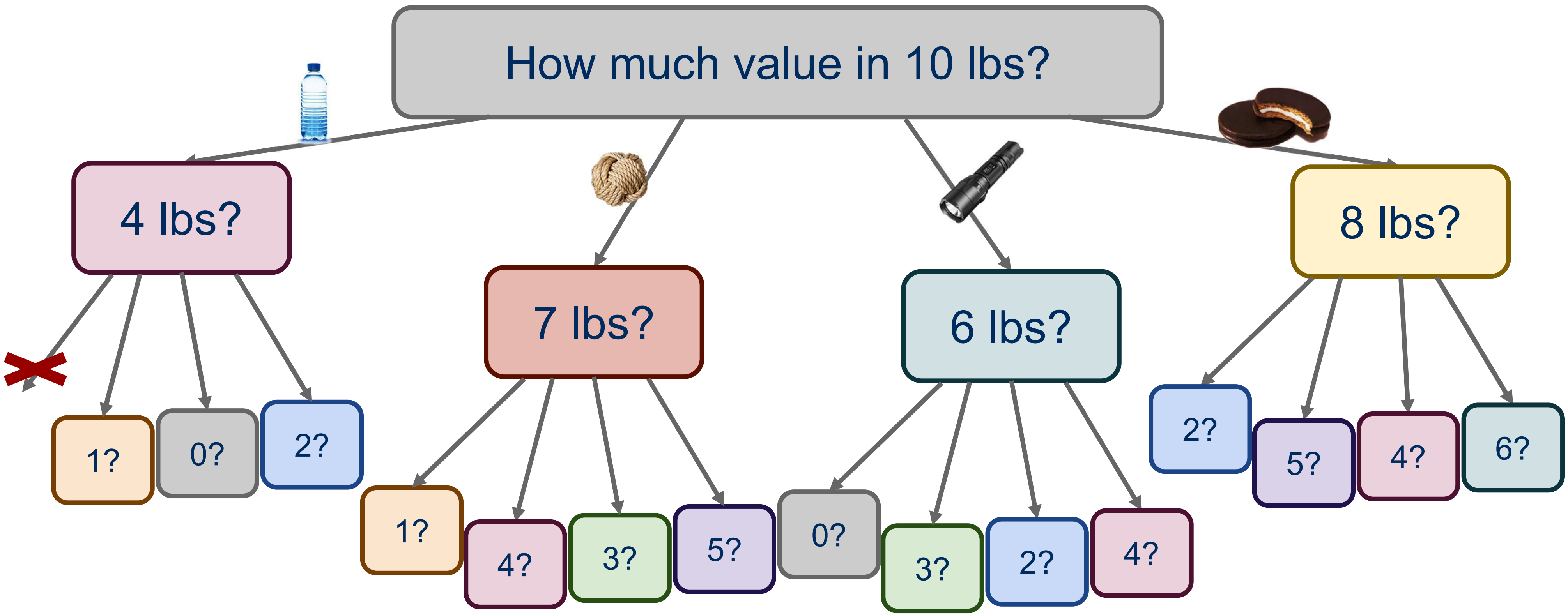
10 lb. capacity



Recursive Solution

weight:	6	3	4	2
value:	30	14	16	9

10 lb. capacity



Bottom-up Solution

				
weight:	6	3	4	2
value:	30	14	16	9

Size:	0	1	2	3	4	5	6	7	8	9	10
Max val:	0	0	9	14	18	23	30	32	39	44	48

Bottom-up solution

```
K[0] = 0
for (l = 1; l <= L; l++) {
      int max = 0;
      for (i = 0; i < n; i++) {
            if (w_i <= l && v_i + K[l - w_i]) > max) {
                  max = v_i + K[l - w_i];
            }
      }
      K[l] = max;
}
```

What would have happened with a *greedy* approach?

- At each step, the algorithm makes the choice that seems to be best at the moment
- Have we seen greedy algorithms already this term?
  - Yes!
    - Building Huffman trees
    - Prim’s, Kruskal’s MST
    - Dijkstra’s Single-Source Shortest Paths

The *greedy algorithm*

- Try adding as many copies of highest value per pound item as possible:
  - Water: 30/6 = 5
  - Rope: 14/3 = 4.66
  - Flashlight: 16/4 = 4
  - Moonpie: 9/2 = 4.5
- Highest value per pound item? Water
  - Can fit 1 with 4 space left over
- Next highest value per pound item? Rope
  - Can fit 1 with 1 space left over
- No room for anything else
- Total value in the 10 lb knapsack?
  - 44
    - Bogus!

But why doesn't the greedy algorithm work for this problem?

- For greedy algorithms to produce optimal results, problems must have two properties:
  - Optimal substructure
    - Optimal solution to a subproblem leads to an optimal solution to the overall problem
  - The greedy choice property
    - Globally optimal solutions can be assembled from locally optimal choices
- Why is optimal substructure not enough?

The bottom-up approach is called dynamic programming!

- Applies to problems with two properties:
  - Optimal substructure
    - Optimal solution to a subproblem leads to an optimal solution to the overall problem
  - Overlapping subproblems
    - Naively, we would need to recompute the same subproblem multiple times
- Greedy Choice Property is not required

Please submit your reflections by using the CourseMIRROR App

If you are having a problem with CourseMIRROR, please send an email to coursemirror.development@gmail.com